

An extensible autonomous reconfiguration framework for complex component-based embedded systems

Johannes Schlatow, Mischa Moestl and Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
{schlatow|moestl|ernst}@ida.ing.tu-bs.de

Abstract—We present a framework based on constraint satisfaction that adds self-integration capabilities to component-based embedded systems by identifying correct compositions of the desired components and their dependencies. This not only allows autonomous integration of additional functionality but can also be extended to ensure that the new configuration does not violate any extra-functional requirements, such as safety or security, imposed by the application domain.

Keywords—component-based; constraint satisfaction; embedded systems; incremental self-integration; software deployment

I. INTRODUCTION

Embedded systems gradually develop from special-purpose towards multi-purpose systems which provide an increasingly diverse functionality. As this functionality is typically developed by several parties and potentially imposes heterogeneous requirements, the integration task becomes increasingly challenging. In the context of component-based systems, integration boils down to composing a subset of available components correctly such that the system provides the requested functionality and also satisfies extra-functional requirements such as safety, availability or security aspects.

Instead of following the more traditional, lab-centric, approach of a more manual integration in conjunction with testing and verification methods for extra-functional requirements, we propose an automated solution that can be integrated into the system itself. This is particularly aligned with systems that are not maintained by a single owner but rather allow several parties to add, remove or modify functionality. For instance, we can observe an increasing interest in in-field updates for maintenance and customisation of safety-critical systems such as automotive vehicles and embedded systems in general [1]. This is also an objective of the *Controlling Concurrent Change (CCC)* project that supported this work. Adding self-reconfiguration capabilities to such a system is desirable as it significantly increases its flexibility but is similarly challenging as the system's safe operation must be ensured without being tested or verified in the lab.

We base our work on systems composed of isolated components interacting via well-defined, explicitly connected interfaces. As embedded systems are subject to resource limitations and typically cannot host all available components at the same time, the integration task consists in selecting a (minimal) subset that provides the desired functionality.

Component-based design methods typically accompany the contract-based specification which formalises the guar-

antees and assumptions that a component requires from its environment. These specifications can address multiple aspects, so-called *design views*, for which a mathematical foundation has been presented in [2]. Furthermore, a design process was proposed and evaluated in industrial case studies that show the real-world applicability of this approach [3].

Based on constraint satisfaction, we propose an extensible software framework for self-integration which is geared towards separation of concerns and therefore facilitates covering a variety of aspects. Based on a formal component model suitable for safety-critical systems, we illustrate how this framework covers and resolves functional dependencies on the component level. In conjunction, this paves the way for in-field updates of complex component-based systems.

We first detail the problem and system model in Section II and III. In Section IV, we then elaborate on the framework's architecture and how the concept of design views integrates with the constraint satisfaction approach. We further formulate the functional constraints on the component level and exemplarily show their constraint encoding in Section V before we finally conclude our findings in Section VII.

II. PROBLEM STATEMENT

When integrating an application component that requires certain service interfaces, the designer first has to find the components that provide these services and likewise integrate those components in order to specify a valid system configuration. This can imply choosing between alternative components that provide syntactically the same service interfaces or deciding whether certain device drivers, protocol stacks or resource multiplexers are required. The solution space for this is typically narrowed down by the availability of the corresponding components. In addition, various functional and extra-functional concerns such as the reliability of a virtual file system or the worst-case latency along a communication path further guide these decisions. We understand that those concerns are addressed within the contract of a component that specifies what properties are required and thus need to be guaranteed by the environment.

Our approach focuses on performing this integration task autonomously and incrementally by respecting all the functional and extra-functional requirements specified in the component contracts (cf. Definition 1). In this context, *autonomously* means that the integration task is performed within the target system and is able to resolve conflicts

between different components and their requirements unsupervised. Moreover, *incrementally* implies that the target system is subject to change, i.e. components may be added, removed or modified after the initial deployment of the system.

Definition 1. The incremental integration problem consists in finding a new system configuration for a given set of components to be deployed and the current system configuration. The found system configuration must be valid, i.e. it must fulfil all the functional and extra-functional requirements specified in the contracts of the deployed components.

III. SYSTEM MODEL

In the CCC project, we aim at providing a contract-based, autonomous integration framework for component-based systems with service-oriented interfaces that follow the principle of least privilege. The formal component model we use as a foundation for this is presented in this section and focuses on the similarities shared between a plethora of existing models [4] to be applicable to a wider range of systems with differing *run-time environments (RTEs)*.

A component-based system is specified by its set of components, their interfaces and the bindings between the interfaces. More precisely, a component may require and/or provide a number of different interfaces that must be bound to a compatible counterpart. This has already been described by [5]. However, an RTE may also impose cardinality restrictions such as the number of bindings as well as optional or conditional bindings. We thus include a notation for these in our component model to demonstrate the compatibility of cardinality and conditional restrictions with our solution.

Let \mathcal{C} denote the set of components, \mathcal{R} the set of requirements and \mathcal{P} the set of provided capabilities. We further distinguish between mandatory and optional requirements, \mathcal{R}_m and \mathcal{R}_o , such that $\mathcal{R} = \mathcal{R}_m \cup \mathcal{R}_o$. Note that requirements correspond to service requirements whereas capabilities correspond to the offered services. Each capability/requirement is owned by exactly one component. The compatibility between capabilities and requirements is subject to the implied service interfaces. We can therefore define the pairwise relationship between components, requirements and capabilities by binary relations:

Definition 2. The relation *requires* $\subseteq \mathcal{C} \times \mathcal{R}_m$ specifies what requirement belongs to which component such that $(c, r) \in \text{requires}$ reads as “component c owns requirement r ”.

Definition 3. The relation *provides* $\subseteq \mathcal{C} \times \mathcal{P}$ denotes what component provides which capabilities such that $(c, p) \in \text{provides}$ reads as “component c provides capability p ”.

Definition 4. The compatibility between the requirements and capabilities is expressed by the relation *satisfiedBy* $\subseteq \mathcal{R} \times \mathcal{P}$ such that $(r, p) \in \text{satisfiedBy}$ reads as “requirement r is satisfied by capability p ”.

Note that the relations *requires* and *provides* additionally classify as *injective* because no requirement/capability can be owned by multiple components. We further extend this model by adding the relation *invokes* and the function ν :

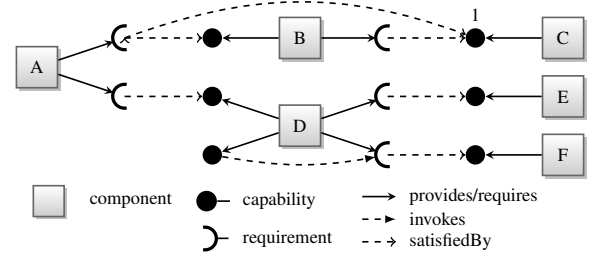


Figure 1. Example of our component model

Definition 5. The relation *invokes* $\subseteq \mathcal{P} \times \mathcal{R}_o$ models requirements that only need to be satisfied if a certain capability of a component is in use such that $(p, r_o) \in \text{invokes}$ reads as “capability p invokes requirement r_o ”.

Definition 6. The function $\nu : \mathcal{P} \rightarrow \mathbb{N}^+$ specifies how many requirements can bind to a particular capability, i.e. how many clients are allowed to connect to the same service.

Note that all these relations are derived from or explicitly stated in the contracts of the components. Based on this, we specify a system configuration as the tuple (α, β) with $\alpha \subseteq \mathcal{C}$ and $\beta \subseteq \text{satisfiedBy}$.

Figure 1 illustrates our component model with an example comprising six components A–F. Component A has two requirements one of which can connect to the capability of component B and C. The other requirement can be connected to a capability of component D. Component B on the other hand not only provides a capability but also has a requirement that is satisfied by the capability of component C. Note that the latter has a cardinality of 1 and B represents a multiplexing component that effectively allows multiple other components to connect to component C. On the other hand, component D provides two capabilities and has two requirements satisfied by the capabilities of component E and F. D therefore represents a protocol-stack component that abstracts the interfaces of E and F. Also note that one of D’s capabilities also invokes one of its requirements, i.e. it only needs to be satisfied as soon as any requirement is connected to the capability.

IV. FRAMEWORK ARCHITECTURE

In [3], a design flow was introduced that separates the different concerns during the lab-based integration task into various design views. While the global problem is to find a feasible system configuration, individual sub-problems are modelled by the respective design view for assessment with an appropriate model. We therefore first adapt the design view concept to our constraint satisfaction approach and, secondly, elaborate on the central *Multi-Change Controller (MCC)*, that controls the autonomous reconfiguration.

Design views: The purpose of a *design view* is the evaluation of possible system configurations under a particular model as to whether certain view-specific requirements are satisfied. As checking every possible configuration is potentially a very expensive computation, we instead define the corresponding solution space by the set of constraints to which each solution adheres.

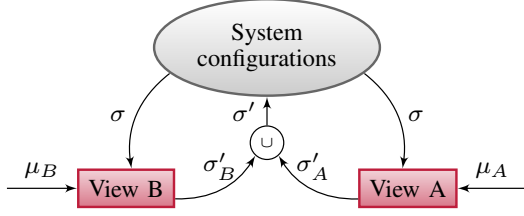


Figure 2. View concept

Definition 7. A view X is a function $V_X : (\sigma, \mu_X) \rightarrow \sigma'_X$ that, given a set of configuration constraints σ and a set of model-specific constraints μ_X , outputs a refined set σ'_X that adheres to μ_X .

For instance, we can define a view A that is concerned about deploying the components across multiple processing resources. This view has a model of the hardware platform along with constraints μ_A regarding the compatibility of the components with the various resources (e.g. instruction set, clock frequency). Based on this additional information, this view evaluates the solution space defined by σ and further restricts it by translating the model-specific constraints μ_A into additional configuration constraints resulting in σ'_A .

Similarly, there are other views that are concerned with other system properties. As illustrated in Figure 2, the configuration constraints of multiple views (here: σ'_A and σ'_B) are combined ($\sigma' = \sigma'_A \cup \sigma'_B$) and fed back into the global solution space of system configurations.

We distinguish four types of constraints: 1) *Necessary and sufficient* constraints tightly restrict the solution space. 2) *Only sufficient* constraints overrestrict the solution space (e.g. by conservative assumptions) thereby cutting off actually feasible system configurations. 3) *Only necessary* constraints underrestrict the solution space and potentially allow unsound solutions (false positives). 4) *Separation* constraints separate unsound solutions from the solution space.

Obviously, type 1 is the most desirable, however, it might be more practical to come up with constraints of type 2 or 3, depending on the formalism of the constraint solver. Type 3 is more desirable but additionally requires a validity check of each found solution in order to detect false positives and (incrementally) separate these by adding type 4 constraints.

Multi-Change Controller: The MCC implements the contract-based autonomous system reconfiguration by resembling the design-view concept mentioned above. For this purpose, it comprises a central constraint solver and coordinates arbitrary design views in an incremental algorithm that is depicted in Figure 3. In the ENC step, all views encode their problem for the constraint solver that tries to obtain a solution in the SLV step. If a view uses only necessary constraints, a so-called *analysis engine* then checks (CHK step) whether an obtained solution can be safely admitted (cf. [6], [7]) or, otherwise, adds (more) separation constraints. The iterative addition of separation constraints also facilitates the integration of so-called *optimisation engines* that search for better solutions in the OPT step and return the optimum once the problem turns infeasible. This further allows integrating existing tools and

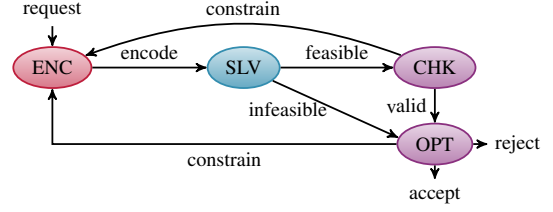


Figure 3. High-level control flow of the controller

established but potentially time-consuming analyses into the framework. The constraint encoding depends on the underlying formalism of the constraint solver and the design-view's model. In the next section, we show an example of the view that adheres to our component model and how its constraints can be encoded for a simple constraint solver.

V. COMPONENT VIEW

In component-based systems, the component view inherently builds the foundation for other views. W.r.t. the reconfiguration of such a system, this view is concerned with the composition of the system components such that all functional dependencies are satisfied. These dependencies are given by the formal component model (Section III) resp. the contracts. Thus, given a query set of components $\kappa \subseteq \mathcal{C}$, the component view shall add constraints so that κ and all their dependencies and connections are instantiated.

For this purpose, we first state the requirements that every system configuration must fulfil: 1) A component must be instantiated if explicitly queried. 2) All mandatory requirements of an instantiated component must be connected to a capability that satisfies the requirement. 3) Every optional requirements of an instantiated component must be connected to a capability that satisfies the requirement if and only if its corresponding capability is used. 4) A component must be instantiated if one of its capabilities is used. 5) A component should not be present if neither explicitly queried nor providing a required capability. 6) A capability p can be used by at most $\nu(p)$ requirements. 7) Already deployed components and their connections shall not be modified during reconfiguration.

These requirements informally represent the constraints on the incremental integration problem and need to be converted into a formalism that can be fed into the particular constraint solver. More precisely, we use a *boolean satisfiability problem (SAT)* solver for which a variety of lightweight implementations is available. In general, an SAT solver expects the constraints, called *clauses*, to be disjunctions (\vee) of boolean variables or their negations (\neg).

For the component view, we introduce these variables:

$$\forall c_i \in \mathcal{C} : q_i = \text{True} \Leftrightarrow c_i \text{ was queried}$$

$$\forall c_i \in \mathcal{C} : c_i = \text{True} \Leftrightarrow c_i \text{ must be instantiated}$$

$$\forall r_j \in \mathcal{R} : r_j = \text{True} \Leftrightarrow r_j \text{ must be satisfied}$$

$$\forall p_k \in \mathcal{P} : p_k = \text{True} \Leftrightarrow p_k \text{ must be provided}$$

$$\forall (r_j, p_k) \in \mathcal{S} : s_{jk} = \text{True} \Leftrightarrow r_j \text{ is connected to } p_k$$

In order to provide an insight into how these requirements are encoded, we take a look at requirement 2. This asserts

that a component is instantiated if one of its capabilities is used and is encoded into the following clauses:

$$\begin{aligned}\forall(r_j, p_k) \in \textit{satisfiedB} : & \neg s_{jk} \vee p_k \\ \forall(c_i, p_k) \in \textit{provides} : & \neg p_k \vee c_i\end{aligned}$$

Note that there exist several methods to encode the cardinality constraints into clauses by adding additional variables [8]. We, however, use the more straightforward binomial encoding that performs reasonably well in our case where the cardinality is often either restricted to 1 or unrestricted.

VI. RELATED WORK

There exists related work that tries to tackle the growing complexity of systems with heterogeneous requirements by introducing model-based design processes. In particular [3] introduced and evaluated a contract-based design process for component-based systems in order to reason separately about the (extra-)functional requirements of different views. It includes functional views – *Data*, *Component*, *Hardware* and *Deployment* – as well as extra-functional, partly domain-specific, views – *Real-time*, *Space-specific*, *Behavioural*, *Dependability* and *Railway-specific*.

On the other hand, [9] presented a layered framework for the integration of heterogeneous systems engineering tools with differing partial models (i.e. views) into an overall model of the system required by the development process. Both approaches, however, focus on a lab-based integration process, which still requires manual decisions.

In the field of constraint-based deployment of components, [10] presented a framework that deploys and manages distributed applications by automatically resolving the mappings and interconnections based on a domain-specific constraint language. Furthermore, [5] formalised the dependency resolution problem for the OSGi component model and presented the corresponding SAT encoding. Yet both approaches only cover single aspects and do not model dependencies on the service level.

There also exists some work with regards to contract-based approaches for self-adaptation of component-based embedded systems. Particularly the FRESCOR project [11] proposed to use service contracts in a layered middleware architecture to manage the adaptation of soft real-time components but does not regard aspects such as safety or availability where hard guarantees are imperative.

VII. CONCLUSION AND FUTURE WORK

In the scope of this work, we developed a framework for autonomous reconfiguration of complex component-based systems that supports the integration of model-based analyses and optimisation objectives. This framework is particularly tailored for the integration into embedded systems in order to conduct admission control of in-field updates and thereby provide adaptivity in critical application domains. It further consolidates well-known concepts and formal methods such as separation of concerns, constraint satisfaction and contract-based design with multiple viewpoints that, in conjunction, facilitate tackling the increasing complexity

and even automate the integration task. In the scope of this work, we focused on the component view that acts as a foundation for other design views by restricting the solution space to correct compositions of components. With the corresponding SAT encoding of the component view, we can already solve the deployment problem in an incremental manner and thus even minimise the number of changes to already instantiated components. Yet performing changes incrementally can quickly lead to a local optimum in the solution space that need to be considered for the long-term evolution of such systems. A case study showed that the practical complexity of such problems is much lower than the theoretically derived bounds. Extra-functional concerns such as end-to-end latencies, however, are subject to future work and may potentially challenge the SAT approach by increasing the problem complexity.

ACKNOWLEDGEMENTS

This work was supported by the DFG Research Unit Controlling Concurrent Change (CCC), funding number FOR 1800. We thank the members of CCC for their support.

REFERENCES

- [1] I. Kuz and Y. Liu, “Extending the capabilities of component models for embedded systems,” in *Intl. Conf. on the Quality of Software-Architectures (QoSA)*, Boston, MA, USA, Jul. 2007.
- [2] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, “Multiple viewpoint contract-based specification and design,” in *FMCO*, ser. Lecture Notes in Computer Science, vol. 5382, 2007, pp. 200–225.
- [3] M. Panunzio and T. Vardanega, “A component-based process with separation of concerns for the development of embedded real-time software systems,” *Journal of Systems and Software*, vol. 96, pp. 105–121, Oct. 2014.
- [4] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron, “A classification framework for software component models,” *IEEE Trans. Softw. Eng.*, vol. 37, pp. 593–615, Sep. 2011.
- [5] G. Jenson, J. Dietrich, and H. W. Guesgen, “An empirical study of the component dependency resolution search space,” in *13th Intl. Symp. on CBSE*, Jan. 2010, pp. 182–199.
- [6] M. Neukirchner, S. Stein, H. Schrom, J. Schlatow, and R. Ernst, “Contract-based dynamic task management for mixed-criticality systems,” in *6th IEEE Intl. Symp. on Industrial Embedded Systems (SIES 11)*, Jun. 2011, pp. 18–27.
- [7] S. Stein, M. Neukirchner, and R. Ernst, “Admission control and self-configuration in the EPOC framework,” in *Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI)*, Jul. 2011.
- [8] A. M. Frisch and P. A. Giannaros, “SAT encodings of the AT-most-k constraint: Some old, some new, some fast, some slow,” 2010.
- [9] M. Gleirscher, D. Ratiu, and B. Schatz, “Incremental integration of heterogeneous systems views,” in *Intl. Conf. on Sys. Eng. and Modeling, 2007. ICSEM 07*, 2007, pp. 50–59.
- [10] A. Dearle, G. N. C. Kirby, and A. J. McCarthy, “A framework for constraint-based deployment and autonomic management of distributed applications,” in *1st International Conference on Autonomic Computing (ICAC 2004)*, 2004, pp. 300–301.
- [11] M. Sojka and Z. Hanzalek, “Modular architecture for real-time contract-based framework,” in *IEEE Intl. Symp. on Industrial Embedded Systems (SIES 09)*, 2009, pp. 66–69.